TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection (Technique Report)

Qiyu Zhuang†, Wei Lu†, Shuang Liu†, Yuxing Chen‡, Xinyue Shi†
Zhanhao Zhao†, Yipeng Sun†, Anqun Pan‡, Xiaoyong Du†
† *Renmin University of China* ‡ *Tencent Inc.*
† *{qyzhuang, lu-wei, shuang.liu, xinyueshi, zhanhaozhao, yipengsun, duyong}@ruc.edu.cn*
‡ *{axingguchen, aaronpan}@tencent.com*

ABSTRACT

Achieving the serializable isolation level, regarded as the gold standard for transaction processing, is costly. Recent studies reveal that adjusting specific query patterns within the workload can still achieve serializability even at lower isolation levels. Nevertheless, these studies typically overlook the trade-off between the performance advantages of lower isolation levels and the overhead required to maintain serializability, potentially leading to suboptimal isolation level choices that fail to maximize performance. In this paper, we present TxnSails, a middle-tier solution designed to achieve serializable scheduling with self-adaptive isolation level selection. First, TxnSails incorporates a unified concurrency control algorithm that achieves serializability at lower isolation levels with minimal overhead. Second, TxnSails employs a deep learning method to characterize the trade-off between the performance benefits and overhead associated with lower isolation levels, thus predicting the optimal isolation level. Finally, TxnSails implements a cross-isolation validation mechanism to ensure serializability during real-time isolation level transitions. Extensive experiments demonstrate that TxnSails outperforms other state-of-the-art solutions by up to 26.7× and PostgreSQL's serializable isolation level by up to 4.8×.

PVLDB Reference Format:
Qiyu Zhuang, Wei Lu, Shuang Liu, Yuxing Chen, Xinyue Shi, Zhanhao Zhao, Yipeng Sun, Anqun Pan, Xiaoyong Du. TxnSails: Achieving Serializable Transaction Scheduling with Self-Adaptive Isolation Level Selection (Technique Report). PVLDB, 18(1): XXX-XXX, 2025. doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbiir/TxnSailsServer.

1 INTRODUCTION

Serializable isolation level (SER) is regarded as the gold standard for transaction processing due to its ability to prevent all forms of anomalies. SER is essential in mission-critical applications, such as banking systems in finance and air traffic control systems in transportation, which require their data to be 100% correct [21]. However, it incurs expensive coordination overhead by configuring the RDBMS to SER [54]. Despite significant efforts to alleviate this overhead [44, 53, 63], maintaining a serial order of transactions to be scheduled remains a fundamental performance bottleneck.

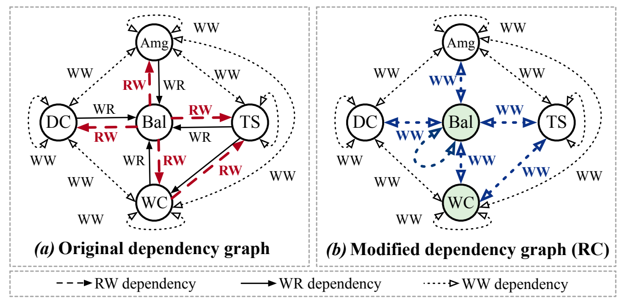


Figure 1: Static dependency graphs of Smallbank

Many studies have explored achieving SER by modifying applications while configuring the RDBMS to a low isolation level [37]. This approach is driven by two key reasons. First, some RDBMSs, such as Oracle 21c, cannot strictly guarantee SER and do not support the in-RDBMS modification, requiring application logic modifications to enforce it. A more comprehensive list of RDBMSs and their supported isolation levels can be found in [13]. Second, RDBMSs typically offer better performance at lower isolation levels, such as read committed (RC) and snapshot isolation (SI), due to their more relaxed ordering requirements. Modifying applications to achieve SER while using a lower isolation level sometimes results in better performance compared to directly setting the RDBMS to SER [11, 12, 54].

The main idea of existing works that can achieve SER under low isolation levels follows three steps: ❶ Build a *static dependency graph* by analyzing the transaction templates, which are the abstraction of transaction logics in real-world applications [54, 55]. In this graph, each template is represented by a vertex, and the dependencies between templates, such as write-write (WW), write-read (WR), or read-write (RW), are depicted as edges. ❷ Configure the RDBMS to a low isolation level and identify *dangerous structures* that are permissible under the low isolation level but prohibited by SER according to the *static dependency graph*. For example, under SI, a dangerous structure is characterized by two consecutive RW dependencies [27, 30], while under RC, a single RW dependency constitutes the dangerous structure [12, 54]. ❸ Eliminate

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

```
Balance(CNAME):
  …
  SELECT BAL FROM Savings
  WHERE CustomerID = :CID;
  …
```

```
Balance(CNAME):
  …
  SELECT BAL FROM Savings
  WHERE CustomerID = :CID FOR UPDATE;
  …
```

Figure 2: Modification of Bal transaction template

dangerous structures by modifying application logic, e.g., promoting reads to writes for certain SQL statements so that the RW dependencies are eliminated, and thus guarantees SER. To achieve this, existing works assume that application logic can be abstracted into transaction templates and that all runtime transactions conform to these templates. This is reasonable as modern applications typically use Object-Relational Mapping (ORM) frameworks [60] to generate structured and repeatable transaction patterns. Therefore, we adopt the same assumption as existing works in this paper, and for interactive or ad hoc transactions not captured by templates, we configure the database to SER to ensure correctness.

Example 1. *Consider the SmallBank benchmark [11], which consists of five transaction templates: Amalgamate (Amg), Balance (Bal), DepositChecking (DC), TransactSavings (TS), and WriteCheck (WC). Suppose the RDBMS is configured to RC. As outlined in step ❶, the benchmark is modeled into a static dependency graph (Figure 1a). At step ❷, five dangerous structures (highlighted by red dashed arrows) are identified, including the dependency from WC to TS and 4 dependencies from Bal to the other four templates. At step ❸, extra writes are introduced to convert RW dependencies to WW dependencies, thus eliminating the dangerous structures. To achieve this, certain "SELECT" statements are modified to "SELECT … FOR UPDATE". Figure 2 illustrates the modification of Bal. For reference, the complete modified dependency graph, based on the original in Figure 1a, is shown in Figure 1b.* □

Thus far, existing studies [14, 15, 20, 28, 29] have proposed promising solutions, enabling RDBMSs to achieve SER by operating at lower isolation levels while modifying specific query patterns within a workload. However, these studies exhibit two major shortcomings. First, the static modification of query patterns is inefficient. These studies alter static SQL statements at the application level, converting certain read operations into write operations. This may result in unnecessary transaction conflicts. For instance, changing read operations to write operations may turn concurrent read-write operations into write-write conflicts in MVCC systems, thus significantly degrading transaction performance. Second, these studies fail to address the key trade-off between the performance gains of lower isolation levels and the overhead needed to maintain SER, making it difficult to choose the optimal isolation level. As shown in Figure 8 of §7, simply configuring the RDBMS to SER can sometimes outperform other methods. Additionally, as workloads evolve, the ideal isolation level may also shift, but existing studies lack the ability to adapt dynamically.

In this paper, we present TxnSails to address the aforementioned shortcomings with three key objectives: ❶ TxnSails efficiently achieves SER under various low isolation levels. ❷ TxnSails dynamically adjusts the optimal isolation level to maximize performance as the workload evolves. ❸ TxnSails is designed to be general and adaptable across various RDBMSs, requiring no modifications to database kernels. To achieve this, TxnSails is implemented as a middle-tier solution to enhance generalizability. However, implementing TxnSails presents three major challenges. First, designing an approach that elevates various isolation levels to SER without introducing additional writes is a complex task. Second, determining the optimal isolation level requires accurately modeling the trade-offs between the performance benefits and serializability overhead associated with lower isolation levels, which is particularly challenging in dynamic workloads. Third, as workloads evolve, the optimal isolation level may need to adapt over time, making it essential to design an efficient and reliable mechanism for transitioning between isolation levels. To address these challenges, we propose the following key techniques.

(1) Efficient middle-tier concurrency control algorithm ensuring SER for each low isolation level (§4.1). We propose a runtime, fine-grained approach that operates on individual transactions rather than transaction templates, ensuring that the execution of transactions meets the requirements of SER. This approach is inspired by the theorem that a schedule is serializable if it does not contain two transactions, T_i and T_j, where T_j commits before T_i, but there is a dependency from T_i to T_j [12]. Building on this theorem, we introduce a unified concurrency control algorithm to ensure SER. The algorithm tracks transactions with their templates involved in specific RW dependency within a static dependency graph. It detects the runtime dependencies and schedules the commit order to align with their dependency order. If necessary, it will abort a transaction to guarantee SER.

(2) Self-adaptive isolation level selection mechanism (§4.2). Rather than directly quantifying a cost model to balance the trade-off between the performance benefits of lower isolation levels and the overhead required to maintain serializability, we employ a learned model leveraging graph neural networks [18] and message-passing techniques [34] to predict the optimal isolation level for a given workload. Our observations reveal that the performance of various isolation levels and the overhead of achieving SER are closely influenced by two critical factors: the data access dependencies between transactions and the data access distribution within transactions. To capture these relationships, we model workload features as a graph, where vertices represent individual transaction features and edges denote data access dependencies between transactions. Building on this insight, we propose a graph-based model for dynamic workloads that predicts the optimal isolation level using real-time workload features. To the best of our knowledge, *TxnSails is the first work to enable self-adaptive isolation level selection for dynamic workloads.*

(3) Cross-isolation validation mechanism that enables efficient transitions and serializable scheduling (§4.3). The optimal isolation level should adapt as the workload evolves. When the RDBMS decides to change the isolation level, new transactions must be executed under this updated isolation level. Although existing approaches can achieve SER when all transactions use a unified low isolation level, they fail to ensure SER when transactions operate under different isolation levels. This is because varying isolation levels can introduce new dangerous structures that may violate the requirements of SER. To address this issue, we identify dangerous structures across different isolation levels and propose a cross-isolation validation mechanism that can prevent the occurrence of these structures during transitions without causing significant system downtime. We prove the correctness of the cross-isolation validation mechanism in §5.2.

We have conducted extensive evaluations on SmallBank [11], TPC-C [7], and YCSB+T [24] benchmarks. The results show that TxnSails can adaptively select the optimal isolation level for dynamic workloads, achieving up to a 26.7× performance improvement over state-of-the-art methods and up to a 4.8× performance boost compared to SER provided by PostgreSQL.

2 PRELIMINARIES

RDBMSs typically offer several isolation levels; in this paper, we focus on the three most commonly used: serializable (SER), snapshot isolation (SI), and read committed (RC). In this section, we first discuss transaction templates. We then present the dangerous structures under SI and RC, respectively. We finally define the vulnerable dependencies that build the foundation of our approach.

2.1 Transaction Templates

A transaction template is an abstraction of application logic that consists of predefined SQL statements with parameter placeholders. Take the Amalgamate template in Example 1 as an example, which facilitates the transfer of funds from one customer to another. It first reads the balances of the checking and savings accounts of customer N_1, then sets them to zero. Finally, it increases the checking balance for N_2 by the sum of N_1's previous balances. In this context, N_1 and N_2 serve as parameter placeholders. This modular structure ensures readability and flexibility, allowing the transaction template to be reused across various contexts. When a customer initiates a transaction at runtime, the application fills the placeholders with actual data. The complete transaction is then executed in the RDBMS, finalizing the business logic.

For better clarity, we use $\mathcal{T}_i$ to denote a transaction template and T_i to denote a transaction generated by $\mathcal{T}_i$.

2.2 Dangerous Structures

The dependencies between two concurrent transactions, T_i and T_j, operating on the same item x, are classified as follows.

- $T_i \xrightarrow{ww} T_j$: T_i writes a version of data item x, and T_j writes a later version of x.
- $T_i \xrightarrow{wr} T_j$: T_i writes a version of data item x, and T_j reads either the version written by T_i or a later version of x.
- $T_i \xrightarrow{rw} T_j$: T_i reads a version of data item x, and T_j writes a later version of x.

Definition 1 (SI dangerous structure [48]). *Under SI, two consecutive RW dependencies: $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ are considered as an SI dangerous structure, where T_i and T_j, T_j and T_k are concurrent transactions, respectively.* □

Definition 2 (RC dangerous structure [12, 32]). *Under RC, an RW dependency: $T_i \xrightarrow{rw} T_j$ is considered as an RC dangerous structure, where T_i and T_j are concurrent transactions.* □

When it comes to transaction templates, the dependencies between two transaction templates, $\mathcal{T}_i$ and $\mathcal{T}_j$, are defined as follows: (1) $\mathcal{T}_i \xrightarrow{ww} \mathcal{T}_j$ if $\mathcal{T}_i$ and $\mathcal{T}_j$ write the same data set (e.g., relation) in sequence; (2) $\mathcal{T}_i \xrightarrow{wr} \mathcal{T}_j$ if $\mathcal{T}_i$ writes and $\mathcal{T}_j$ reads the same data set in sequence; (3) $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j$ if $\mathcal{T}_i$ reads and $\mathcal{T}_j$ writes the same data set in sequence.

Definition 3 (Static SI dangerous structure [10, 19]). *In a static dependency graph, two consecutive edges $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j, \mathcal{T}_j \xrightarrow{rw} \mathcal{T}_k$ are deemed to constitute a static SI dangerous structure.* □

Definition 4 (Static RC dangerous structure [12, 55]). *In a static dependency graph, an edge $\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j$ is deemed to constitute a static RC dangerous structure.* □

Theorem 2.1 ([9]). *If a static dependency graph contains no SI (resp. RC) static dangerous structures, then scheduling the transactions generated by the corresponding transaction templates achieves SER when the RDBMS is configured to SI (resp. RC).* □

Theorem 2.1 serves as the foundation for existing approaches to achieving SER while the RDBMS is configured to SI/RC. However, these approaches are static and coarse-grained, leading to the incorrect identification of many non-cyclic schedules. This, in turn, causes a significant number of unnecessary transaction rollbacks.

2.3 Vulnerable Dependency

Definition 5 (Static vulnerable dependency). *The static vulnerable dependency is defined as $\mathcal{T}_j \xrightarrow{rw} \mathcal{T}_k$ in chain $\mathcal{T}_i \xrightarrow{rw} \boxed{\mathcal{T}_j \xrightarrow{rw} \mathcal{T}_k}$ under SI, and $\boxed{\mathcal{T}_i \xrightarrow{rw} \mathcal{T}_j}$ under RC, respectively.* □

Definition 6 (Vulnerable dependency). *The vulnerable dependency is defined as $T_j \xrightarrow{rw} T_k$ in chain $T_i \xrightarrow{rw} \boxed{T_j \xrightarrow{rw} T_k}$ under SI, and $\boxed{T_i \xrightarrow{rw} T_j}$ under RC, respectively.* □

Theorem 2.2 ([12]). *For any vulnerable dependency $T_i \xrightarrow{rw} T_j$, if T_i commits before T_j, then the scheduling achieves SER.* □

Theorem 2.2 forms the basis of our dynamic, fine-grained approach to achieving serializable scheduling. Compared to existing approaches, our approach neither introduces unnecessary writes nor misjudges cyclic schedules, thus preventing unwarranted transaction rollbacks.

3 OVERVIEW OF TxnSails

TxnSails works in the middle tier between the application tier and the database tier, designed to ❶ ensure SER when transactions operate under a low isolation level without introducing additional writes; ❷ select the optimal isolation level for dynamic workloads adaptively; ❸ constantly keep SER during the isolation level transition. An overview of TxnSails is depicted in Figure 3. It comprises three main components: *Analyzer*, *Executor*, and *Adapter*.

Analyzer. *Analyzer* provides *template interfaces* for template registration and analysis. Before TxnSails executes any transaction from the application, *Analyzer* builds the static dependency graph for the transaction templates and identifies all the static vulnerable dependencies for each low isolation level according to Definition 5. It then sends the static vulnerable dependencies to *Executor*.

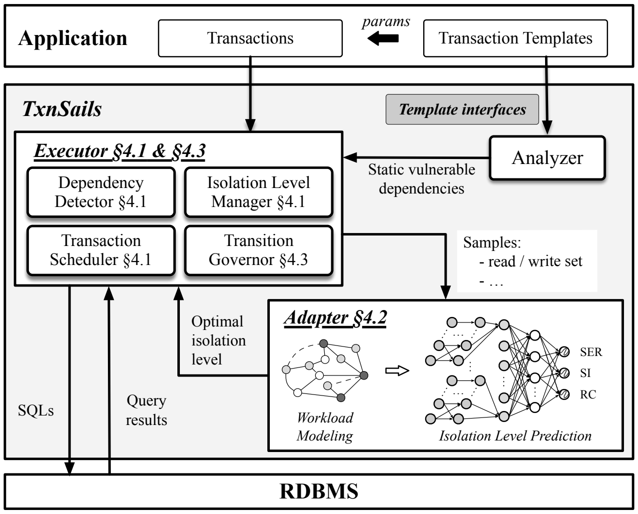


Figure 3: An overview of TxnSails

Executor. *Executor* ensures SER when transactions operate either at a single low isolation level or during the isolation level transition. There are four core modules: *Isolation Level Manager (ILM)*, *Dependency Detector (DD)*, *Transaction Scheduler (TS)*, and *Transition Governor (TG)*. (1) ILM stores the static vulnerable dependencies, and before any transaction T starts, it identifies whether T involves any static vulnerable dependencies. If the template of T does not involve static vulnerable dependencies, *Executor* sends T directly to the RDBMS for execution; otherwise, ILM triggers DD that identifies vulnerable dependencies of T. (2) DD monitors the reads and writes of T, detecting any runtime vulnerable dependencies between T and other transactions. If T is involved in any vulnerable dependencies, TS is triggered. (3) TS attempts to ensure that the commit and vulnerable dependency orders remain consistent between T and other transactions. If the consistency cannot be guaranteed, T is blocked or aborted; otherwise, T proceeds to commit. (4) TG ensures SER during the transition between two isolation levels. It follows a new corollary, which extends Theorem 2.2 to any two transactions, T_i and T_j, executing under different isolation levels. The proof of correctness during the isolation level transition is detailed in §5.2.

Adapter. *Adapter* models the trade-off between performance benefits and serializability overhead under lower isolation levels. It predicts the optimal isolation level when the workloads evolve. Initially, a dedicated thread is introduced to continuously sample aborted/committed transactions using Monte Carlo sampling [68], capturing the read/write data items. After collecting a batch of transaction samples, *Adapter* predicts the optimal isolation level for future workloads based on the characteristics of the batch. The prediction process consists of two steps: *Workload Modeling (WM)* and *Isolation Level Prediction (ILP)*. WM extracts performance-related features and models the workload as a graph. In this graph, each vertex represents a runtime transaction, with its features capturing the transaction context, such as the number of data items in the read and write sets. Each edge represents an RW or WW operation dependency between transactions. Following WM, ILP embeds the workload graph into a high-dimension vector using graph neural network [18] and message passing techniques [34], and then translates the vector into three possible labels: RC, SI, or SER. The label with the highest value, as determined by our model, indicates the most efficient isolation level.

For reference, the detailed implementation of the interfaces and the three core components are provided in §6.

4 DESIGN OF TxnSails

In this section, we provide the detailed design of TxnSails. We first introduce the middle-tier concurrency control mechanism that achieves serializable scheduling when the RDBMS is configured to a low isolation level (§4.1). Then, we present a self-adaptive isolation level selection approach, which can predict the optimal future isolation level (§4.2). Lastly, we introduce the cross-isolation validation mechanism that ensures serializable scheduling during the isolation level transition (§4.3).

4.1 Middle-tier Concurrency Control

Existing approaches ensure serializability at low isolation levels by statically introducing additional write operations. However, these approaches reduce concurrency and increase overhead. To overcome these limitations, TxnSails introduces a middle-tier concurrency control algorithm, which dynamically validates runtime dependencies and schedules their commit order. In particular, TxnSails focuses exclusively on vulnerable dependencies identified by the *Analyzer* and employs a lightweight validation mechanism to mitigate overhead further.

4.1.1 Transaction lifecycle. The lifecycle of transactions in the middle tier is divided into three phases: execution, validation, and commit phases. (1) In the execution phase, TxnSails establishes a database connection with a specific isolation level, which is not adjusted until the transaction is committed or aborted. Following the RDBMS transaction execution, TxnSails stores the read/write data items in the thread-local buffer that may induce the vulnerable dependencies; (2) In the validation phase, TxnSails acquires validation locks for data items stored in the buffer. Then, it detects the dependencies among them and aims to schedule the commit order consistent with the identified dependency order. A more detailed description of the validation phase will be given in §4.1.2; (3) In the commit phase, TxnSails applies modifications to the database and subsequently releases the validation locks.

4.1.2 Validation phase. TxnSails performs two key tasks in the validation phase: (1) detecting vulnerable dependencies; (2) scheduling the commit order consistent with the dependency order. To achieve this, we explicitly add a *version* column to the schema, which is incremented after every update. We trace the dependency orders by comparing the versions of data items. It is worth noting that the explicit version column is introduced for the convenience of description. It can be achieved by extracting version information from the database without modifying the schema. For example, PostgreSQL maintains the version information via the ***ctid***. Algorithm 1 shows the detailed algorithm.

For both RC and SI levels, we detect vulnerable dependencies based on those defined in Definition 6. During validation, the transaction first requests *Shared* locks for items in the read set and *Exclusive* locks for items in the write set (lines 2-7). Specifically, before

Algorithm 1: Middle-tier concurrency control algorithm

```
1  Function Validate(T, conn):
       Input: T, transaction requiring validation;
              conn, a connection under RC
       // Acquire the validation locks on data items
2      for r in T.vread_set ∪ T.vwrite_set do
3          res := TryValidationLock(r.key, T.tid, r.type)
4          while res is WAIT do
5              res := TryValidationLock(r.key, T.tid, r.type)
6          if res is ERROR then
7              return ERROR
       // Check the version of data items in the read set
8      for r in T.vread_set do
9          version := 0
10         entry := VLT.get_lock_entry(r.key)
11         if entry.version > 0 then
               // get the latest version from version cache
12             version := entry.version
13         else
               // fetch the latest version from DBMS
14             version := conn.get_version(r.key)
15             entry.version := version
16         if version is not r.version then
17             return ERROR
18     return SUCCESS
19 Function Commit(T, sess):
       Input: sess, session for transaction execution;
20     sess.commit(T)
21     for r in T.vwrite_set do
22         entry := VLT.get_lock_entry(r.key)
23         entry.version = r.version
24     for r in T.vread_set ∪ T.vwrite_set do
25         ReleaseValidateLock(r.key)
```

transaction T_i commits, the validation phase is performed in two key steps:

(1) T_i checks each data item in its write set to detect the RW dependencies from T_i's concurrent transactions (e.g., T_j) to T_i. We achieve this via the validation locks. If any lock request fails, indicating T_j exists, an RW dependency is detected. In such a case, the failed lock request should be appended to the corresponding lock's waitlist, making T_i wait until T_j commits, ensuring consistency between dependency and commit orders. If no concurrent transactions in the validation phase are reading the same item, T_i proceeds to commit and create a new data version.

(2) T_i checks each data item in its read sets to detect the RW dependencies from T_i to T_i's concurrent transactions. Specifically, it compares the version of each read item in the thread-local buffer with the latest version in the database (lines 9-15). If a newer version is found, indicating an RW dependency from the current transaction T_i to a committed transaction, say T_j, then T_i is aborted to ensure the consistency of commit and dependency orders (lines 16-17). Moreover, comparing data item versions in the local buffer with the latest versions in the RDBMS can introduce additional

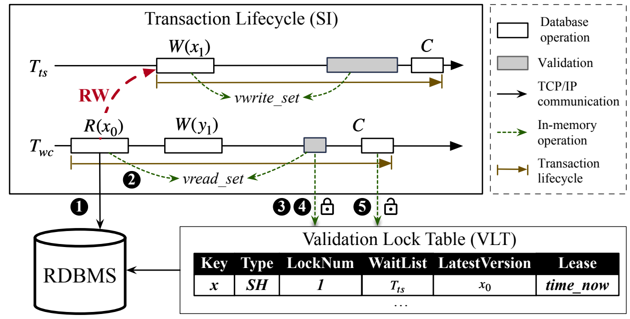


Figure 4: Transaction processing in TxnSails

interactions between the database middleware and the underlying database, imposing overhead on system and network resources. To alleviate this burden, TxnSails employs a caching mechanism in the middle-tier memory to store the latest versions of data items. By enabling rapid retrieval of the most recent versions, this approach significantly reduces validation overhead and enhances overall efficiency.

In the above steps, we ensure the commit order in the middle tier is consistent with the dependency order. Subsequently, we schedule the actual commit order in the RDBMS consistent with the commit order in the middle tier. Middle-tier consistency is achieved by aborting or blocking transactions when detecting a vulnerable dependency. We ensure that the RDBMS layer consistency is achieved by releasing validation locks only after the transaction has been successfully committed in the database (lines 24-25). Based on this, if two concurrent transactions access the same data item (with one writing and the other reading or writing), they cannot both enter the validation phase simultaneously. One transaction must complete validation and commit before the other can proceed, ensuring a correct and consistent commit order in the RDBMS.

To enable efficient and accurate validation, TxnSails leverages a validation lock table (VLT) to maintain metadata for each data item. Each data item is assigned a hash value computed using the collision-resistant hash function $\mathcal{H}$, and a corresponding entry is stored in VLT. Data items with the same hash value are stored in the same bucket and organized as a linked list. When an entry with key x is accessed, TxnSails first determines the appropriate bucket using $\mathcal{H}(x)$ and then traverses it to locate the specific entry. Each hash entry e comprises five fields: (1) $e.Type$, the type of locks acquired, which can be *None*, *Shared (SH)*, and *Exclusive (EX)*; (2) $e.LockNum$, the number of currently held locks; (3) $e.WaitList$, a list of transactions waiting to acquire locks; (4) $e.LatestVersion$, the most recent committed version of the data item; and (5) $e.Lease$, the timestamp indicating the garbage collection time.

Example 2. *Take Figure 4, which provides a concise depiction of transaction processing, as an example. Recall that there exists a static vulnerable dependency* $\mathcal{T}_{wc} \xrightarrow{rw} \mathcal{T}_{ts}$ *in Smallbank when the RDBMS is set to SI (Figure 1). Thus, it is necessary to detect the read operation of* T_{wc} *and the write operation of* T_{ts}*. In the execution phase, after the RDBMS execution (❶),* T_{wc} *stores the data item x in its vread_set and* T_{ts} *stores x in its vwrite_set (❷). In the validation phase of* T_{wc}*, it acquires the shared validation lock on x (❸) and retrieves the latest* *version of x from either VLT or the RDBMS (❹). While in the validation phase of T_{ts}, it requests the exclusive validation lock on x and is blocked until T_{wc} releases the lock. Finally, in the commit phase, T_{wc} releases the validation lock on x (❺). This ensures that the commit order of the two transactions is consistent with the dependency order, thereby guaranteeing SER when they operate under SI.* □

4.1.3 Discussion. To optimize memory usage, TxnSails incorporates an efficient garbage collection algorithm to evict cold entries. Upon accessing an entry e in the VLT, TxnSails updates $e.Lease$ with a future timestamp, then traverses the corresponding bucket to identify and remove outdated, unused entries where *Lease* falls behind than the current timestamp and $Type$ is *None*. Additionally, to prevent entries in infrequently accessed buckets from persisting indefinitely, a background thread periodically scans these long-unused buckets and evicts outdated entries, ensuring efficient memory management across the system. For complex queries on primary keys, TxnSails uses multi-grained validation locks (such as interval or table locks) to detect predicate dependencies, drawing on techniques similar to SIREAD locks in PostgreSQL and gap locks [43]. When TxnSails cannot easily infer target items in complex queries (e.g., with predicates on non-primary keys), it defaults to table-level locks or configures the underlying database to use SER for correctness.

4.2 Self-adaptive Isolation Level Selection

Selecting optimal isolation levels for all transactions in a workload while maintaining SER is challenging, as we need to balance the extra serializability overhead and performance benefits in different isolation levels. Inspired by existing approaches that effectively conduct workload prediction using neural networks [46, 61, 66], we propose a neural-network-based isolation level prediction approach, which predicts the future optimal isolation level based on the current workload features. Specifically, TxnSails adopts transaction dependency graphs to capture workload features and adopts a graph classification model to perform self-adaptive prediction.

4.2.1 Graph construction. To extract the complex features of concurrent transactions, TxnSails proposes a graph-structured workload model, which is composed of three matrixes: a vertex matrix V, an edge index matrix E, and an edge attribute matrix A. Formally, a workload graph is defined as $G = (V, E, A)$, where each row in A represents the feature vector of an operator, each entry e_{ij} in E signifies the relationship between v_i and v_j, and each row in A represents the feature vector of an edge.

TxnSails dynamically constructs the runtime workload graph by sampling transactions adhering to Monte Carlo sampling. Each transaction in the batch is mapped to a vertex v_i, and its feature vector V_i is generated by extracting the number of data items in its read set and write set. For each vertex pair (v_i, v_j), if a data dependency exists between them, i.e., their read and write sets intersect, TxnSails adds an edge entry e_{ij} into the edge index matrix E. For each edge e_{ij}, TxnSails extracts the data dependency type and the involved relations to generate its attribute $A_{e_{ij}}$. The data dependency type for e_{ij} can be either RR, RW/WR, or WW. For example, if two transactions' write sets intersect, there is a WW dependency between them. Given that the number of relations and dependency

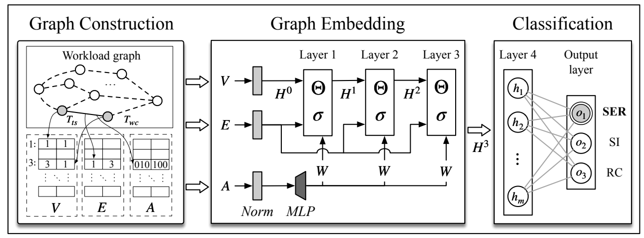


Figure 5: Graph-based isolation level selection model

types are fixed, one-hot encoding is employed to represent these two features within the attribute matrix.

4.2.2 Graph embedding and isolation prediction. Predicting the optimal isolation strategy for the future workload using the constructed graph-structure model $G = (V, E, A)$ is challenging due to its complex structures and dynamic and high-dimensional features, which require capturing both local and global dependencies. Heuristic methods rely on manually crafted rules that lack generalizability, while traditional machine learning models are deficient in leveraging relational information encoded in the vertexes and edges, losing critical structural context. To address these challenges, we use a graph classification model that learns graph-level representations by aggregating node features through multiple layers of neural network-based convolutions.

As shown in Figure 5, our graph model comprises two parts. First, we use a *Graph Embedding Network* to learn and aggregate both vertex and edge features, producing node-level embedded matrix H that encodes the local structure and attribute information of the graph. Second, to predict the optimal isolation strategy for the workload, we use a *Graph Classification Network* that learns the mapping from the embedded matrix H to perform the end-to-end graph classification to predict the optimal isolation strategy.

The *Graph Embedding Network* is constructed with a three-layer architecture, where each layer applies a convolution operation to update node representations. This process integrates node and edge features through a dynamic aggregation mechanism [31, 67]. At each layer, an edge network maps the input edge features into higher-dimensional convolution kernels via a multi-layer perception (MLP), as shown in Eq.(1). This mapping dynamically transforms edge attributes into weights, which are then used during the node aggregation step. The convolution operation produces updated node embeddings for each node v_i using Eq. 1, where $\mathcal{N}(v_i)$ represents the neighbors of node v_i, $W^{(l)}_{e_{ij}}$ is the edge-specific weight, and σ denotes the active function (i.e., ReLU). Through this layer-wise propagation, the embedding module produces H, a set of node-level embeddings that encode the graph information.

$$\begin{cases} W^{(l)} = f^{(l)}(A) = MLP(A) \\ H^{(l)}_{v_i} = \sigma\left(\max_{v_j \in \mathcal{N}(v_i)} \left(W^{(l)}_{e_{ij}} \cdot H^{(l-1)}_{v_j}\right)\right) \end{cases} \tag{1}$$

The *Graph Classification Network* takes the node embeddings H as inputs and passes them through two fully connected layers. The first layer applies a ReLU activation function to enhance nonlinearity. The second layer implements a softmax activation function and outputs a three-dimensional vector, with each field representing the probability of the isolation level being optimal.

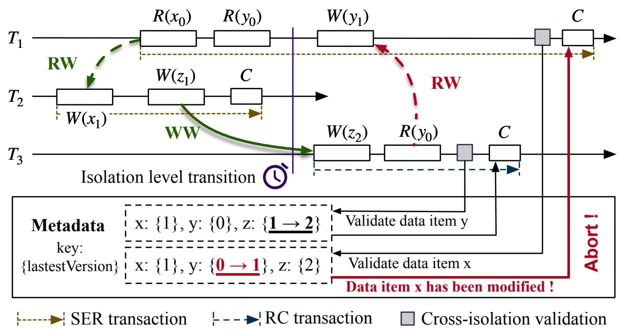


Figure 6: Cross-isolation validation

4.2.3 Data collection and labeling. Our modeling approach is somewhat general and not specifically designed for any particular workloads. However, in practice, we train the model separately for each type of benchmark for efficiency considerations. Taking YCSB+T as an example, we generate lots of random workloads with varying read/write ratios and key distributions. Each workload is executed under each isolation level for 10 seconds, with sampling intervals of 1 second, and the optimal isolation level is labeled based on throughput. We follow the same process for data collection and labeling in Smallbank and TPC-C.

4.2.4 Model training. In TxnSails, we train the embedding and prediction network together and use cross-entropy loss for multi-class classification. Backpropagation involves calculating the gradients of the loss function concerning the parameters of the graph model. First, the gradient is computed for the output layer. Then, using the chain rule, these gradients are propagated backward through the whole network, updating the parameters of each layer. For embedding layers, this process includes computing gradients for both vertex features and transformation matrices derived from edge attributes. The model training overhead after the transaction template change is insignificant because our model parameters are 550k. The model can be retrained asynchronously, and during the retraining period, it does not affect transaction execution.

4.3 Cross-isolation Validation

If the predicted optimal isolation level changes, TxnSails will adapt from the previous isolation level I_{old} to the optimal isolation level I_{new}. We design a cross-isolation validation mechanism to guarantee SER during the isolation level transition.

Example 3. *Figure 6 illustrates non-serializable scheduling during the transition from SER to RC after T_2 commits, making T_1 and T_2 operate under SER while T_3 operates under RC. In this scenario, T_1 is expected to be aborted to ensure SER. However, existing RDBMSs do not handle dependencies between transactions under different isolation levels, allowing T_1 to commit successfully, leading to non-serializable scheduling. Note that when transactions T_1, T_2, and T_3 are all executed under SER, the concurrency control in RDBMS prevents such non-serializable scheduling.* □

We need to explicitly consider the situations of cross-isolation transitions to ensure the correct transaction execution during the process. A straightforward approach is to wait for all transactions to complete under the previous isolation level before making the transition. In the example above, this would mean blocking T_3 until T_1 commits. However, it can result in prolonged system downtime, especially when there are long-running uncommitted transactions. Another possible approach is to abort these uncommitted transactions and retry them after the transition, which leads to a high abort rate. To mitigate these negative impacts, TxnSails employs a cross-isolation validation (CIV) mechanism that ensures serializability and allows for non-blocking transaction execution without a significant increase in aborts. Specifically, we extend the vulnerable dependency under the single isolation level in Definition 6 to the cross-isolation vulnerable dependency, defined as follows:

Definition 7 (Cross-isolation vulnerable dependency). *The cross-isolation vulnerable dependency is defined as $T_j \xrightarrow{rw} T_k$ in the dependency chain $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ where T_j commits after the transition starts.* □

Based on Definition 7, we extend Theorem 2.2, which is defined for the single isolation level, to derive Corollary 1 for cross-isolation transitions, and prove its correctness in §5.2.

Corollary 1. *During the transition, if every cross-isolation vulnerable dependency $T_j \xrightarrow{rw} T_k$ in a transaction scheduling satisfies that T_j commits before T_k, this scheduling is serializable.* □

Based on Corollary 1, we implement our CIV mechanism by detecting all cross-isolation vulnerable dependencies during the isolation-level transition and ensuring the consistency of the commit and dependency orders. The CIV mechanism includes three steps. (1) When the system transitions from the current isolation I_{old} to the optimal isolation level I_{new}, the middle tier blocks new transactions from entering the validation phase until all transactions that have entered the validation phase before the transition commit or abort. Importantly, we only block transactions to enter the validation phase. Transactions can execute normally without blocking. (2) After that, the transaction that has completed the execution phase enters the cross-isolation validation phase. During the cross-isolation validation phase, transactions request validation locks according to the stricter locking method of either I_{old} or I_{new} to ensure that all cross-isolation vulnerable dependencies can be detected. For example, when transitioning from SI to RC, the transaction in the cross-isolation validation phase requests validation locks following RC's validation locking method, regardless of whether it is executed under SI or RC. (3) After acquiring validation locks, transaction T_i first detects vulnerable dependencies of its original isolation level. Then, it detects cross-isolation vulnerable dependencies by checking whether a committed transaction modifies its read set (using the same detection method as that in §4.1). If such modifications are detected, T_i is aborted to ensure the consistency of the commit and dependency orders.

Once all transactions executed under I_{old} are committed or aborted, the transition process ends. Then, transactions do not need to undergo the cross-isolation validation.

According to the CIV mechanism, before transaction T_1 can commit, TxnSails validates its validation read set (x_0 and y_0). Since T_1 has created a new version x_1 and commits before T_1, TxnSails would abort T_1 to ensure serializability.

Specifically, we prove that the scheduling during the transition between RC and SI is serializable if they perform the concurrency control in §4.1 (detailed in §5.2).

5 SERIALIZABILITY AND RECOVERY

In this section, we first prove the serializability of TxnSails's scheduling in the single-isolation level and cross-isolation level categories in § 5.1 and § 5.2, respectively. Finally, we present the failure recovery strategy in § 5.3.

5.1 Serializability under Low Isolation Levels

Non-serializable scheduling under each low isolation level accommodates certain specific vulnerable dependencies. According to Theorem 2.2, a necessary condition for non-serializability is the presence of inconsistent dependencies and commit orders among these vulnerable dependencies. TxnSails identifies static vulnerable dependencies from the transaction templates and ensures that, in transactions involving these dependencies, the commit order aligns with the dependency order as specified in Algorithm 1. This approach maintains SER even when the RDBMS is configured to low isolation levels.

5.2 Serializability under Cross-isolation Levels

In this subsection, we prove the serializability during isolation transitions after implementing our CIV mechanism in §4.3. The proof can be reduced into two steps: (1) we first prove the correctness of Corollary 1; (2) then we prove that our CIV mechanism can detect each cross-isolation vulnerable dependency, $T_j \xrightarrow{rw} T_k$, in Definition 7 and ensure the consistency between the commit order and the dependency order between T_j and T_k, thus ensuring serializability during isolation transitions.

5.2.1 Correctness of Corollary 1. We prove the correctness of Corollary 1 by proving the correctness of its contrapositive. Corollary 1 can be formalized using Eq.(2) where ct_{T_j} represents the T_j's commit time and ts denotes the isolation transition's start time.

$$\forall (T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k \wedge ct_{T_j} > ts \wedge ct_{T_j} < ct_{T_k}) \Rightarrow SER \tag{2}$$

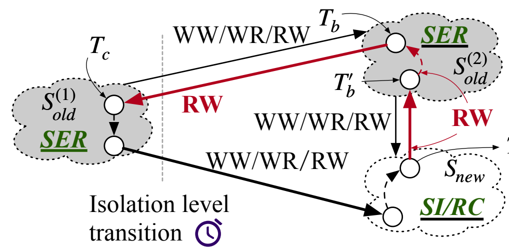
Its contrapositive can be deduced using Eq.(3). For clarity, we use α to denote $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k \wedge ct_{T_j} > ts$.

$$\begin{aligned}
\neg SER &\Rightarrow \neg\forall(\alpha \wedge ct_{T_j} < ct_{T_k}) \Rightarrow \exists\neg(\alpha \wedge ct_{T_j} < ct_{T_k}) \\
&\Rightarrow \exists(\neg\alpha \vee \neg ct_{T_j} < ct_{T_k}) \\
&\Rightarrow \exists(\neg\alpha \vee (\neg ct_{T_j} < ct_{T_k} \wedge (\alpha \vee \neg\alpha))) \\
&\Rightarrow \exists(\neg\alpha \vee (\neg ct_{T_j} < ct_{T_k} \wedge \neg\alpha) \vee (\neg ct_{T_j} < ct_{T_k} \wedge \alpha)) \\
&\Rightarrow \exists((\neg\alpha \wedge (True \vee \neg ct_{T_j} < ct_{T_k})) \vee (\neg ct_{T_j} < ct_{T_k} \wedge \alpha)) \\
&\Rightarrow \exists(\neg\alpha \vee (\neg ct_{T_j} < ct_{T_k} \wedge \alpha))
\end{aligned} \tag{3}$$

Since any witness of $\neg ct_{T_j} < ct_{T_k} \wedge \alpha$ automatically satisties $\neg\alpha \vee (\neg ct_{T_j} < ct_{T_k} \wedge \alpha)$, the proof of Corollary 1's contrapositive can be reduced to proving $\neg SER \Rightarrow \exists(\neg ct_{T_j} < ct_{T_k} \wedge \alpha)$, which is demonstrated in Proposition 1.

Proposition 1. *If a non-serializable schedule occurs during the isolation transition, there must exist a cross-isolation vulnerable dependency, $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_j commits after the transition starts, and T_k commits before T_j.* □

We then prove the correctness of Proposition 1 based on the following two steps. ❶ If there is a non-serializable scheduling during



Dependency orders between transaction sets

Figure 7: Transition from SER to SI/RC

the transition, there exists at least one anomaly structure defined in Definition 8. ❷ If there is a non-serializable scheduling during the transition, there exists at least one anomaly structure that satisfies T_j commits after the transition starts. Due to space constraints, a detailed proof is available in our technical report [2].

Definition 8 (Anomaly Structure). *The anomaly structure during transition is defined as $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_j operates under SER and T_k commits before T_j.*

The proof is based on Theorem 5.1.

Theorem 5.1. *If there is a WR/WW data dependency from any two transactions T_i to T_j, T_i must be committed before T_j starts.*

❶ We first prove the existence of the anomaly structure in Definition 8. As is well known, if there exists a non-serializable scheduling, then the dependency graph formed by the transactions involved in that scheduling must contain a dependency cycle [8]. Without loss of generality, we refer to the first transaction committed in this cycle as T_0. Then, we label the dependency chain involving T_0 and its two predecessor transactions as $T_2 \xrightarrow{D_1} T_1 \xrightarrow{D_2} T_0$. Notably, T_2 and T_0 may refer to the same transaction if the dependency cycle only contains two transactions, while this scenario also conforms to the following proof. We first prove that both D_1 and D_2 are RW dependencies. Since T_0 commits first, therefore, T_1 cannot commit before T_0 begins, making D_2 must be an RW dependency according to Theorem 5.1. Similarly, D_1 must also be an RW dependency. As shown in Figure 7a, since $T_1 \xrightarrow{rw} T_0$, T_1 must begin before T_0 commits. Also, T_2 commits after T_0 commits; therefore, T_2 cannot commit before T_1 begins, making D_1 must also be an RW dependency according to Theorem 5.1. Further, since T_0 commits before T_1, we can derive the existence of two consecutive RW dependencies $T_i(T_2) \xrightarrow{rw} T_j(T_1) \xrightarrow{rw} T_k(T_0)$ where T_k commits before T_j.

Then we prove that T_1 (corresponding to T_j in Definition 8) must execute under SER. First, T_1 can not operate under RC. If T_1 operates under RC, then the RW dependency between T_1 and T_0 conforms the vulnerable dependency definition under RC (Definition 6), making the commit order and dependency order between T_1 and T_0 must be consistent according to the concurrency control in §4.1. Second, T_1 cannot operate under SI. If T_1 operates under SI, the RW dependency between T_1 and T_0 conforms to the vulnerable dependency definition under SI (Definition 6), therefore, it can also be detected by the concurrency control in §4.1. Hence, if non-serializable scheduling exists, **there must be two consecutive RW dependencies, $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_k commits before T_j and T_j operates under SER.**

Since the non-serializable scheduling must contain a transaction that operates under SER, **the transition between RC and SI is serializable under the concurrency control in §4.1**.

❷ We then prove that the existence of the cross-isolation vulnerable dependency $T_j \xrightarrow{rw} T_k$, where T_k commits before T_j and T_j commits after the transition starts. We demonstrate the proof under two cases as follows.

Transition from SI/RC to SER. Given that T_j in the anomaly structure in Definition 8 operates under the new isolation level (SER), it must commit after the transition starts.

Transition from SER to SI/RC. For clarity, we categorize the transactions during the transition into three discrete sets:

- $S_{old}^{(1)}$: The set of transactions under I_{old} and have been committed when the transition starts.
- $S_{old}^{(2)}$: The set of transactions that operate under I_{old} and commit after the transition starts.
- S_{new}: The set of transactions that start after the transition occurs and operate under I_{new}.

We then prove that there must exist at least one anomaly structure stated as Definition 8 where $T_j \in S_{old}^{(2)}$.

Given Theorem 5.1, we can derive the dependency edges between transaction sets and demonstrate it in Figure 7.

(1) The red arrow from S_{new} to $S_{old}^{(2)}$ shows that data dependencies from S_{new} to $S_{old}^{(2)}$ can only be RW dependencies due to the transactions in S_{new} cannot commit before transactions in $S_{old}^{(2)}$ start.
(2) The red arrow within $S_{old}^{(2)}$ represents that dependencies between transactions within $S_{old}^{(2)}$ can only be RW dependencies because they are concurrent transactions.
(3) The red arrow from $S_{old}^{(2)}$ to $S_{old}^{(1)}$ shows that dependencies from transactions in $S_{old}^{(2)}$ to those in $S_{old}^{(1)}$ must be RW dependencies due to transactions in $S_{old}^{(2)}$ cannot commit before transactions in $S_{old}^{(1)}$ start.
(4) There is no arrow from S_{new} to $S_{old}^{(1)}$ due to all transactions in $S_{old}^{(1)}$ commit before any transaction in S_{new} starts.

Given the dependency order between transactions, we can observe that the dependency cycles of non-serializable cross-isolation scheduling only include two cases: (a) scheduling involves only transactions in $S_{old}^{(2)}$ and S_{new}; (b) scheduling involves transactions in $S_{old}^{(1)}$, $S_{old}^{(2)}$ and S_{new}.

In the first case, all transactions in the dependency cycle are committed after the transition has occurred. According to Definition 8, there exists a transaction chain $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ in the dependency cycle, where T_j operates under SER. Since T_j operates under SER and commits after the transition, $T_j \in S_{old}^{(2)}$.

In the second case, based on Definition 8, there exists a transaction chain $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$ forming part of the dependency cycle, where T_j operates under SER. Since T_j operates under SER, $T_j \notin S_{new}$. Then we prove that it is impossible for every T_j involved in all anomaly structures to belong to $S_{old}^{(1)}$. We prove this by contraposition. Suppose there is a dependency cycle C, and consider any transaction $T \in S_{old}^{(2)} \cap C$. Such a transaction T can not be the middle transaction T_j in the anomaly structure $T_i \xrightarrow{rw} T_j \xrightarrow{rw} T_k$, where T_k commits before T_j. As a result, for any dependency in C that goes from transactions in $S_{old}^{(2)}$ to those in $S_{old}^{(1)}$, the incoming dependency to the transaction in $S_{old}^{(2)}$ must be a WW/WR dependency. Referring to the example in Figure 7, there is a dependency from T_b to T_c, where T_b's predecessor should belong to $S_{old}^{(1)}$. Since this dependency from T_b to T_c can represent any dependency from transactions in $S_{old}^{(2)}$ to those in $S_{old}^{(1)}$, it follows that no consecutive chain can exist from transactions in S_{new} to those in $S_{old}^{(1)}$. Therefore, the dependency cycle C cannot exist, contradicting our initial assumption. Hence, there must be at least one transaction $T_j \in T_{old}^{(2)}$ that appears as the middle transaction in an anomaly structure.

Given the above proofs, we can derive that if there is a non-serializable scheduling, then there must be at least one anomaly structure in Definition 8 where $T_j \in S_{old}^{(2)}$. In other words, at least one T_j commits after the transition starts.

5.2.2 *Serializability during transition.*

Then we prove that our CIV mechanism can detect all vulnerable dependencies described in Definition 7, thus ensuring the serializability. In the CIV mechanism, we employ validation for every transaction that is not committed after the transition starts, ensuring it is not involved in a cross-isolation vulnerable dependency and maintaining consistent dependency and commit orders. With this validation, we can ensure that there does not exist any cross-isolation vulnerable dependency during the transition, thus ensuring the serializability.

5.3 Failure Recovery

The system incorporates a robust failure recovery mechanism to ensure data consistency and service availability. For TxnSails failures, the RDBMS first automatically rolls back uncommitted transactions because the connection between the RDBMS and TxnSails is lost. Next, the TxnSails is automatically restarted, reconnected to the RDBMS, and then continues the normal execution. For RDBMS failures, the RDBMS is restarted and recovered to a consistent state using its recovery algorithm (e.g., ARIES [47]). For both failures, both components are restarted, relying on the RDBMS's recovery mechanism to ensure consistency.

6 IMPLEMENTATION

We implement TxnSails from scratch using Java and Python, comprising approximately 6,000 lines of Java code and 500 lines of Python code. TxnSails is publicly available via [2]. Implemented outside the database kernel, TxnSails can seamlessly integrate with any RDBMS that offers the isolation levels defined in [8, 48], enhancing performance and ensuring SER.

Interfaces. Applications interact with TxnSails via predefined interfaces, as detailed in Table 1. Applications first register transaction templates by *register* interface, which parses the sql to extract

Table 1: Interfaces of TxnSails

Transaction template interfaces	
register(template_name, sql)	Register each sql with the template names.
analysis()	Analyze and identify static vulnerable dependencies in low isolation levels.

operation types and relations. Then, *analysis* interface is used to identify static vulnerable dependencies under low isolation levels.
Analyzer. We implement *SDGBuilder* class that takes transaction templates as input and constructs a static dependency graph. The graph is then passed to *CycleFinder* class to detect cycles based on the characteristics defined in Theorem 2.1. Finally, it identifies transaction templates with static vulnerable dependencies and stores the results in a *MetaWorker* instance.
Executor. During execution, TxnSails automatically analyzes runtime transactions and identifies which template transactions belong to. It invokes *SQLRewrite()* function to rewrite queries, selecting the appropriate record version if its template is involved in static vulnerable dependencies. Additionally, we implement a critical data structure, *ValidationMetaTable*, which is initialized before any transactions are received to perform middle-tier validation in both single- and cross-isolation scenarios. Organized as a hash table, each bucket in the table represents a list of *ValidationMeta* entries, including *validation lock*, *latest version*, and *lease* information. A dedicated thread handles garbage collection of expired meta entries by comparing the *lease* with the system's real-time clock. Moreover, we implement the *WAIT-DIE* strategy within the *ValidationMetaTable* to prevent deadlocks.
Adapter. We first implement *TransactionCollector* class that collects the read and write sets for transactions. Then, we implement a *RDGBuilder* class to build the runtime dependency graph. Finally, *Adapter* is implemented with the aid of *torch_geometric*, taking the runtime dependency graph as input and outputting the optimal isolation level. To ensure cross-platform compatibility and efficiency, the Python and Java components communicate via *sockets*.
Integration with Apache Shardingsphere [40] The *Analyzer* and *Adapter* can be directly integrated as external modules. Specifically, we extend the *PostgreSQLMultiStatementsHandler* to collect validation information from execution results and store it in the *ConnectionSession*, based on static vulnerable dependencies. Additionally, we modify the *TransactionBackendHandler* to perform validation using the metadata gathered during execution, ensuring consistency between dependency and commit order.

7 EVALUATIONS

In this section, we evaluate TxnSails's performance compared to state-of-the-art solutions. Our goal is to validate two critical aspects empirically: (1) TxnSails's effectiveness in adaptively selecting the appropriate isolation level for dynamic workloads (§7.2); and (2) TxnSails's performance superiority over state-of-the-art solutions across a variety of scenarios (§7.3).

7.1 Setup

We run our database and clients on two separate in-cluster servers with an Intel(R) Xeon(R) Platinum 8361HC CPU @ 2.60GHz processor, which includes 24 physical cores, 64 GB DRAM, and 500 GB SSD. The operating system is CentOS Linux release 7.9.

7.1.1 Default configuration. We utilize BenchBase [26] as our benchmark simulator, which is deployed on the client server. By default, the experiments are conducted using 128 client terminals. We deployed PostgreSQL 15.2 [6] as the default database engine, which employs MVCC to implement three distinct isolation levels: Read Committed (RC), Snapshot isolation (SI), and Serializable (SER) (by SSI [19]). Under RC, the system can read the most recently committed version, while both SI and SER maintain a view of the data as it existed at the start of the transaction, thereby observing the committed version from that point in time. To prevent dirty writes, write locks are enforced at all isolation levels. For our database configuration, we allocated a buffer pool size of 24GB, limited the maximum number of connections to 2000, and established a lock wait timeout of 100 ms. To eliminate network-related effects, both TxnSails and PostgreSQL were deployed on another server.

7.1.2 Baselines. To ensure a fair comparison, we implemented existing approaches within the BenchBase framework.
Baselines within the database. We evaluated concurrency control algorithms supported natively by RDBMSs, specifically those associated with lower isolation levels that can achieve SER:

*(1) & (2) Native concurrency control mechanisms in RDBMSs (**SER** and **SI**).* These approaches execute workloads configured at the SER or SI levels. For instance, TPC-C achieves serializable scheduling under SI, while SmallBank requires SER for serializability.
Baselines outside the database kernel. We also evaluated external strategies that transform RW dependencies into WW dependencies to eliminate static dangerous structures.

*(3) & (4) Promotion (**RC+Promotion**[55], **SI+Promotion** [11]).* This strategy converts read operations into write operations by promoting *SELECT* statements with non-modifying *UPDATE* statements [11]. These modifications are applied at RC and SI levels, referred to as RC+Promotion and SI+Promotion, respectively.

*(5) & (6) Conflict materialization (**RC+ELM** [12], **SI+ELM** [11]).* This approach employs an external lock manager (ELM) and introduces additional write operations on the ELM to ensure serializable scheduling. It is applied at both RC and SI levels, referred to as RC+ELM and SI+ELM, respectively.

We incorporate two simple isolation level selection methods: a rule-based method and a Bayesian model-based method [33], denoted as **TxnSails-Rule** and **TxnSails-Bayesian**, respectively. TxnSails-Rule relies on the write/read ratio (wr), using SI for $wr < 0.2$, SER for $0.2 \leq wr \leq 0.4$, and RC for $wr > 0.4$. TxnSails-Bayesian uses a 4-dimensional feature vector, including the number of reads (ro), writes (wo), RW dependencies (rw), and WW dependencies (ww), to represent each workload.

For each strategy, we adopted the most effective variant as identified in prior work [12]. For example, under the Promotion strategy in the SmallBank benchmark with SI, modifying the *WriteCheck* template rather than the *Balance* template yielded superior performance.

Finally, we evaluate the middle-tier concurrency control in §4.1 at both RC and SI levels without self-adaptive isolation level selection, denoted as **TxnSails-RC** and **TxnSails-SI**, respectively.

7.1.3 Benchmarks. Three benchmarks are conducted as follows.

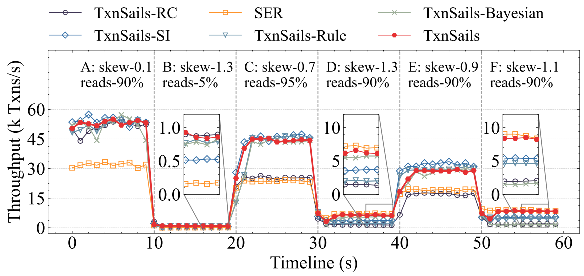
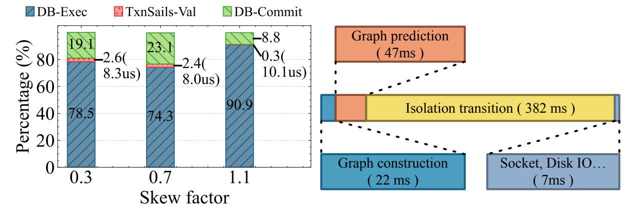
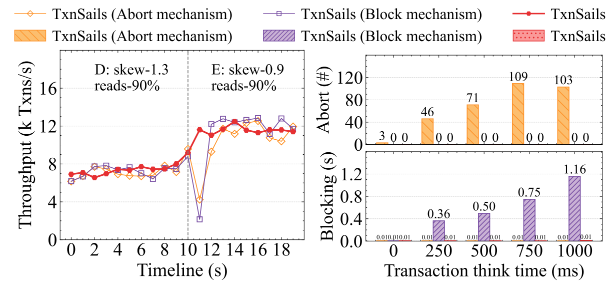


Figure 8: Workload shifting by YCSB



(a) Transaction breakdown **(b) Transition breakdown**

Figure 9: Breakdown analysis by YCSB



(a) Transition from D to E **(b) Performance metrics**

Figure 10: Comparison of transition mechanisms - YCSB

SmallBank [11]. This benchmark populates the database with 400k accounts, each having associated checking and savings accounts. Transactions are selected by each client using a uniform distribution. To simulate transactional access skew, we employ a Zipfian distribution with a default *skew factor* of 0.7.

YCSB+T [25]. This benchmark generates synthetic workloads emulating large-scale Internet applications. In our setup, the *usertable* consists of 10 million records, each 1KB in size, totaling 10GB. The *skew factor*, set by default to 0.7, controls the distribution of accessed data items, with higher values increasing data contention. Each default transaction involves 10 operations, with a 90% probability of being a read and a 10% probability of being a write.

TPC-C [7]. We use the TPC-C benchmark, which modifies the templates to convert all predicate reads into key-based accesses according to our baseline [54]. It includes five transaction templates: NewOrder, Payment, OrderStatus, Delivery, and StockLevel. Our tests use 32 warehouses, with a total data size of 3.2 GB. Following previous works [35, 62], we exclude user data errors that cause about 1% of NewOrder transactions to abort.

7.2 Ablation Study

In this part, we evaluate the effectiveness of the self-adaptive isolation level selection and isolation transition in TxnSails.

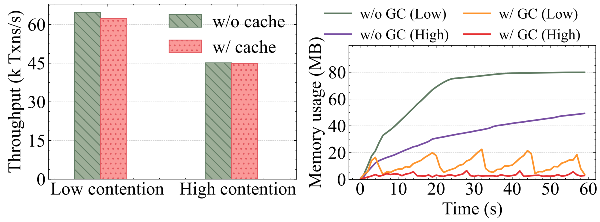
7.2.1 Self-adaptive isolation level selection. We first evaluate self-adaptive isolation level selection by varying the workload every 10 seconds across six distinct scenarios. The experimental results are illustrated in Figure 8. We sample the workload at 1-second intervals. The results demonstrate that different isolation levels perform variably under different workloads: SI performs well in low-skew scenarios (A, C, E), SER is more effective in high-skew scenarios with a lower percentage of write operations (D, F), and RC excels in high skew scenarios with a high percentage of writes (B). Across all tested dynamic scenarios, TxnSails successfully adapts to the optimal isolation level. Specifically, the graph learning model in TxnSails identifies that SI is suitable for scenarios with fewer conflicts due to its higher concurrency and lower data access overhead (i.e., one-time timestamp acquisition). Conversely, RC is ideal for scenarios with higher conflict rates and more write operations, as it efficiently handles concurrent writes (SI aborts concurrent writes, while RC allows them to commit). Compared to an application directly run on RDBMS at the SER level, the performance of TxnSails when selecting SER is slightly reduced by 4.3%. This overhead arises from two aspects: (1) TxnSails requires selecting *version* and another localhost message delivery ; (2) TxnSails needs to sample transactions to predict the optimal isolation level, even though this is an asynchronous task. The rule-based and Bayesian model-based methods achieve correct predictions in some cases; however, they fail to make optimal decisions in scenarios such as workloads D and F, due to their limited modeling capacity compared to the neural network.

7.2.2 Validation analysis. This part evaluates the validation efficiency in both single-isolation and cross-isolation scenarios.

Single-isolation level validation. We evaluate the single-level validation cost under skew factors of 0.3, 0.7, and 1.1, respectively. The average breakdown is depicted in Figure 9a. The validation cost remains relatively stable, decreasing from 2.6% to 0.3% of the transaction lifecycle as contention increases. This suggests that the middle-tier concurrency control proposed in §4.1 does not significantly affect normal execution.

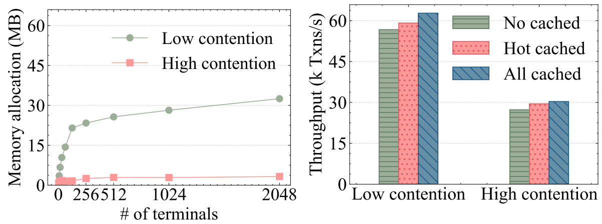
Cross-isolation level validation. We evaluate TxnSails with YCSB using the various transition mechanisms mentioned in §4.3. We first evaluate the transition of the workload from D to E with a "think time" of 1s, as illustrated in Figure 10a. TxnSails minimizes the impact of isolation level transitions by avoiding active block time or aborts, maintaining serializable scheduling. To further compare mechanisms, we vary the "think time" parameter (Figure 10b). Increased think time raises transaction latency and leads to more aborts under the abort strategy, while the blocking strategy incurs longer blocking times. In contrast, the cross-isolation validation mechanism outperforms both, reducing transaction aborts and blocking time while delivering performance improvements of up to 2.7× and 5.4×, respectively.

7.2.3 Overhead analysis. We first evaluate the metadata-induced memory overhead, as shown in Figure 12. With 10GB of YCSB data, TxnSails uses under 50MB of memory while achieving a 10% performance gain. The cache hit ratio is 39.8% under low contention and 68.4% under high contention, indicating that TxnSails



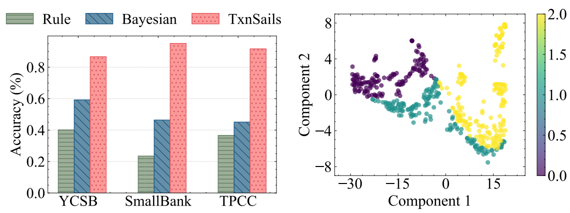
(a) Performance **(b) Memory usage**

Figure 11: Impact of garbage collection - YCSB



(a) Memory usage **(b) Performance**

Figure 12: Impact of Metadata - YCSB



(a) Accuracy **(b) Extracted features**

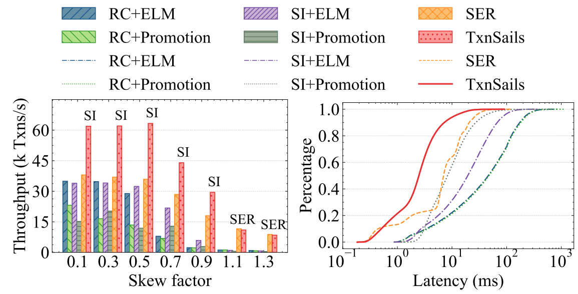
Figure 13: Model training metrics - YCSB

efficiently caches frequently accessed tuples, especially when contention is high. Then, we evaluate the overhead of our garbage collection strategy. As illustrated in Figure 11, the garbage collection mechanism incurs only slight performance overheads, 3.61% under low contention and 0.6% under high contention, while reducing memory usage by 84.7% and 92.0%, respectively.

7.2.4 Graph model: construction, training, and prediction. Figure 9b illustrates the overhead of workload transition, which takes approximately 450 milliseconds (ms). Specifically, graph construction and prediction require 22 ms and 47 ms, respectively, while over 80% of the time is spent on transition, from initiating the transition to all connections adopting the new isolation level, closely tied to the longest transaction execution latency. Notably, the prediction in Figure 8 is inaccurate for 1 or 2 seconds at the 30-second and 50-second marks due to the sampling transactions from the previous workload during the transition. However, the model successfully transitions to the optimal isolation level in subsequent prediction cycles. The overhead of the learned model is minimal, with less than a 2.5% difference in throughput between using the graph model and not using it.

We compare the accuracy of three prediction methods in different benchmarks. As is shown in Figure 13a, the neural network method consistently achieves the highest accuracy across all benchmarks, outperforming the rule-based and Bayesian models by up to 27.5%, 48.7%, and 46.6%, respectively. The two simpler methods perform slightly better on YCSB, which has less workload semantics and fewer transaction templates. The superior performance of the neural network is due to its ability to capture complex relationships in the dependency graph, whereas the other two simpler approaches rely on irreversibly compressed or aggregated features, limiting their effectiveness. To visualize the high-dimensional vectors produced by our model, we use t-SNE [3] for nonlinear dimensionality reduction, mapping them into two dimensions and plotting them with their true labels in Figure 13b. Most workloads are accurately distinguished, with errors primarily occurring at the boundaries between isolation levels, where performance similarities can lead to incorrect predictions that do not significantly impact overall performance.

Summary. One isolation level does not fits all workloads. In low-skew scenarios, SI outperforms RC; in high-skew scenarios with fewer writes, SER is the most effective; and in high-skew scenarios with intensive writes, RC is more suitable. TxnSails effectively guarantees SER at lower levels and efficiently adapts isolation levels to optimize performance for dynamic workloads using the proposed fast isolation level transition technique.



(a) Performance **(b) Analysis of latency CDF**

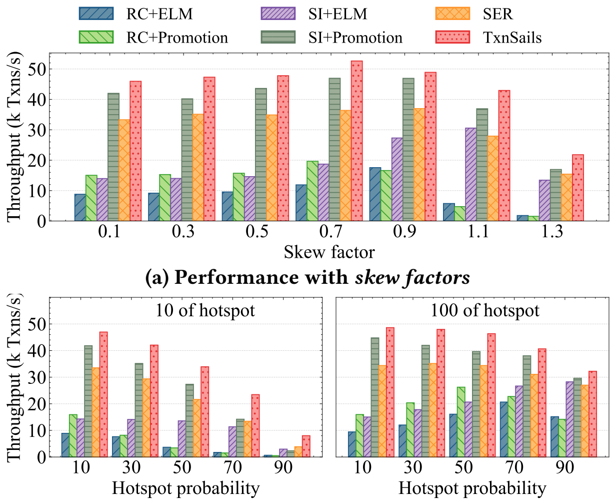
Figure 14: Impact of data contention - YCSB

7.3 Comparision to State-of-the-art Solutions

We evaluate TxnSails against state-of-the-art solutions that use **external lock manager (ELM)** [11, 12] and **Promotion** [11, 55] over workloads by YCSB, SmallBank, and TPC-C benchmarks.

7.3.1 Impact of data contention. This part studies the impact of data contention by varying the *skew factor* (SF) and by varying *hotspot probability* and the number of hotspots to simulate different data contention in YCSB and SmallBank, respectively.

In YCSB, TxnSails outperforms other solutions by up to 22.7× and is 1.9× better than the second-best solution due to lightweight validation without workload modification, thus higher concurrency as depicted in Figure 14a. In this case, SOTA solutions can not beat SER as they introduce additional write operations in YCSB workloads that restrict concurrency. In high contention scenarios (SF>0.9), validation costs outweigh the benefits of using a lower isolation level, triggering TxnSails to transition to the SER level and perform slightly (<5%) lower than SER due to TxnSails overhead. In low contention scenarios (SF<0.9), TxnSails adopts SI due to low validation overhead. Meanwhile, in low contention scenarios, the ELM approaches yield better efficiency than the Promotion ones, as lock conflicts are low with external locks. We further analyze the latency distribution with the skew factor of 0.9 using cumulative distribution function (CDF) plots, as shown in Figure 14b. In all scenarios, TxnSails reduces the latency of transactions.



(a) Performance with *skew factors*

(b) Performance with fixed number of hotspots

Figure 15: Impact of data contention - SmallBank

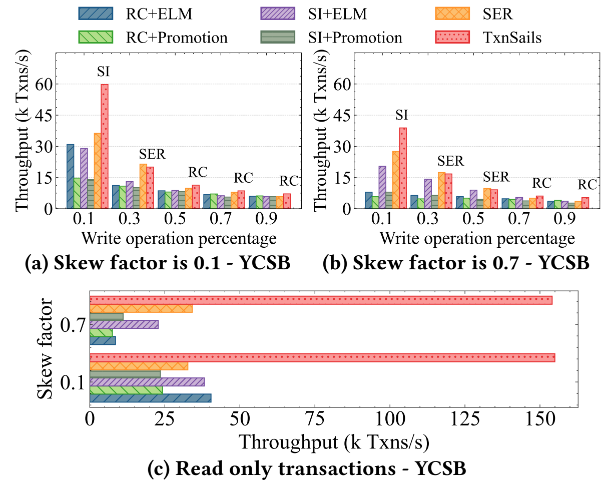
In SmallBank, TxnSails consistently outperforms other solutions by up to 15.27× improvement and 2.06× better than the second-best solution, as depicted in Figure 15. This time, SI+Promotion can outperform SER. The reason is that, unlike YCSB, SmallBank validates only a small portion of read and write operations. As the skew factor increases, TxnSails maintains its advantage over SER by employing SI level. As hotspot size increases, the reduced conflicts between transactions make the performance advantage of TxnSails less pronounced.

7.3.2 Impact of write/read ratios. This part evaluates the performance of varying the percentage of write operations over YCSB, using the *skew factors* of 0.1 and 0.7. In read-write scenarios in Figure 16a and 16b, TxnSails can outperform second best solutions up to 1.7×. As the percentage of write operations increases, the performance gap narrows as verification overhead at lower isolation levels increases. TxnSails transitions from using SI to SER and finally to RC, as the FCW [21] strategy increases the abort rate in scenarios with a high percentage of writes.

We also evaluate the performance in read-only scenarios in Figure 16c. TxnSails achieves performance up to 4.6× higher than SER and up to 20.4× higher than others. TxnSails adopts to SI level as its in-memory validation is nearly costless and rarely fails. Other solutions convert read operations to write operations, thereby restricting concurrency. This also highlights that when a database is configured to be SER, there is a significant performance loss compared to SI, even with read-only scenarios.

7.3.3 Impact of templates percentages. In complex workloads like SmallBank and TPC-C, only certain transaction templates lead to data anomalies, so modifying these templates can ensure serializability under low isolation levels. This part compares different solutions by varying the percentage of critical transaction templates.

In SmallBank, we evaluate the proportions of the read-only *Balance* and write transaction *WriteCheck*, as shown in Figure 17. As the proportion of *Balance* transactions increases, performance improves; however, RC+ELM and RC+Promotion introduce additional writes in *Balance*, leading to increased WW conflicts. In contrast, SI+ELM and SI+Promotion perform better since they do



(a) Skew factor is 0.1 - YCSB

(b) Skew factor is 0.7 - YCSB

(c) Read only transactions - YCSB

Figure 16: Impact of write/read ratio - YCSB

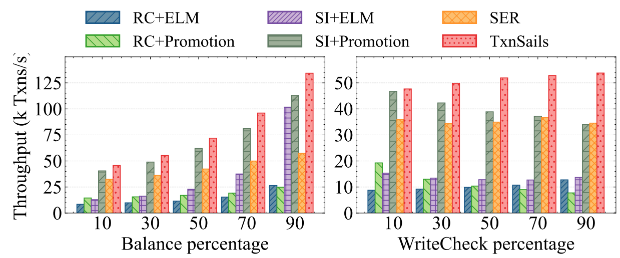


Figure 17: Impact of templates percentage - SmallBank

not modify read-only *Balance* transactions. TxnSails-RC must detect RW dependencies from *Balance*, increasing overhead as their proportion rises. Thus, TxnSails transitions to SI in this scenario, achieving up to a 6.2× performance gain over RC+ELM and RC+Promotion. Conversely, as the proportion of *WriteCheck* transactions increases, concurrency decreases, leading to worse performance for SI+ELM and SI+Promotion. However, TxnSails's performance advantage becomes more pronounced as it maintains consistent commit and dependency orders through validation without modifying the workload. At 90% *WriteCheck* transactions, TxnSails improves performance by 58.1% compared to SI+Promotion and achieves 2.3× the performance of SER.

TPC-C can execute serializable under SI, eliminating the need for validation in SI. The critical *NewOrder* and *Payment* transactions require modifications by RC+ELM and RC+Promotion, increasing write conflicts on *NewOrder*, resulting in a performance disadvantage compared to TxnSails, which can achieve up to 2.3× their performance. Due to high contention on the warehouse relation, validation overheads are generally higher, except when the proportion of *NewOrder* is 0.1, where TxnSails shows a 10.7% improvement over SI. In other scenarios, TxnSails adapts to SI. *Payment* transactions are more write-intensive, prompting TxnSails to adapts to RC, which achieves up to 40.6% performance improvement over SI. Compared to other solutions, TxnSails achieves up to 53.5% performance improvement.

7.3.4 Generality and scalability. This part evaluates the generality and scalability of TxnSails using MySQL and Neon [5]. Specifically, we apply TxnSails to low isolation levels in MySQL and

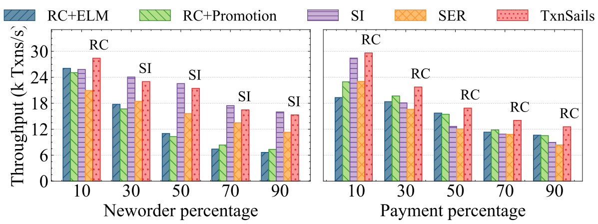
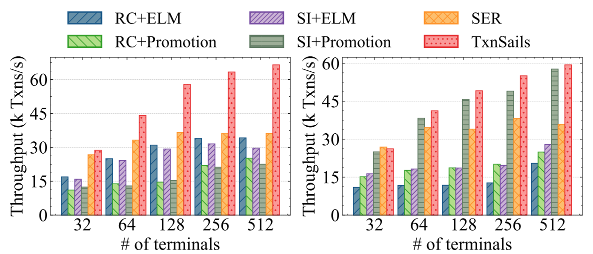


Figure 18: Impact of templates percentage - TPC-C



(a) Performance - YCSB **(b) Performance - SmallBank**

Figure 19: Impact of client terminal numbers

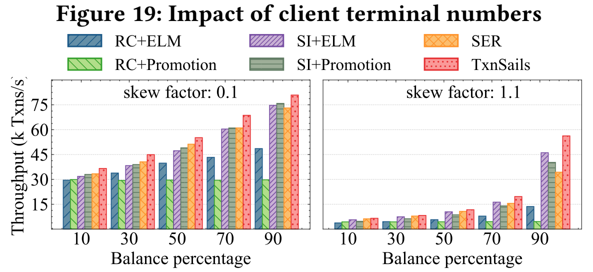
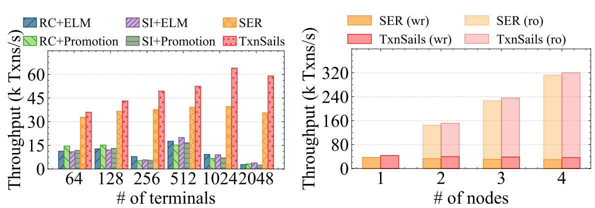


Figure 20: Extended evaluations on MySQL - SmallBank



(a) Scale up **(b) Scale out**

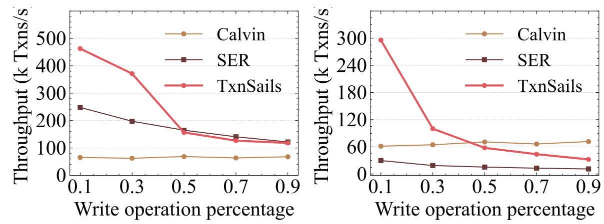
Figure 21: Scalability in Neon - YCSB

conduct experiments using the SmallBank benchmark. As shown in Figure 20, TxnSails consistently achieves the highest throughput on MySQL. In particular, TxnSails outperforms the second-best solution by up to 12.5% under low contention and 22.1% under high contention workloads. Furthermore, TxnSails surpasses other solutions by up to 12.36×. These results highlight the generality of TxnSails across diverse database systems.

To evaluate the scalability, we conduct experiments on Neon, an elastically scalable database, from both scale-up and scale-out perspectives. As shown in Figure 21a, we vary the number of client terminals on a single read-write node from 32 to 2048. The results show that TxnSails consistently outperforms other solutions. TxnSails achieves up to 1.65× performance improvement at 2048 terminals. As shown in Figure 21b, we increase the number of read-only nodes. Here, TxnSails also consistently surpasses the baseline Neon database, confirming its great scale-out scalability. It is worth noting that, since read-only nodes only provide SI, TxnSails does not add extra concurrency control overhead on these nodes.

Table 2: Integerate TxnSails into Apache shardingsphere

Approach	Throughput (k Txn/s)	Median Latency (ms)	p99 Latency (ms)
SSP+TxnSails	32.3	3.97	21.66
SSP	28.4	4.76	49.12



(a) Low contention ($sf = 0.1$) **(b) High contention ($sf = 1.1$)**

Figure 22: Compare against deterministic approaches

Consequently, the performance gains become less pronounced as the number of read-only nodes increases.

Additionally, we evaluate the scalability under various numbers of client terminals, as shown in Figure 19. TxnSails outperforms other solutions by up to 3.96× and 4.21× by YCSB and SmallBank, respectively. As client terminals increase, TxnSails consistently outperforms, primarily due to its small overhead associated with external concurrency control management at a lower isolation level. Interestingly, SER outperforms most other solutions, as these solutions often introduce additional write operations that restrict concurrency and require updating extensive locking information. The notable exception occurs in the SmallBank workload, where the SI+Promotion method surpasses SER. This improvement can be largely attributed to the modification of a limited number of transaction templates within SmallBank.

7.3.5 Integration into Shardingsphere. We extend Shardingsphere's transaction manager to support serializable isolation level when the underlying data source is set to low isolation levels. Moreover, we evaluate the TxnSails in Shardingsphere with the underlying data source PostgreSQL via YCSB workload with default configuration; the result is illustrated in Table 2. After integrating TxnSails, SSP's performance improves 13.7%, the median latency reduces 16.5%, and the maximum latency reduces up to 55.9%. TxnSails can provide better performance and lower tail latency with a low isolation level.

7.3.6 Comparison with deterministic concurrency control algorithm. In this section, we implement TxnSails into Deneva [36], an open-source distributed benchmarking system with Calvin ready. The serializable isolation level is implemented using Serializable Snapshot Isolation (SSI), consistent with PostgreSQL. As shown in Figure 22, TxnSails outperforms Calvin by up to 6.84× under low-contention workloads. Conversely, Calvin exhibits better performance in high-contention scenarios when the write ratio exceeds 30%. Given that most OLTP workloads tend to be read-heavy with relatively low write intensity, TxnSails offers a practical advantage in such environments, requiring fewer client application logic assumptions.

Summary. Current research often limits concurrency and scalability in a coarse-grained manner by replacing read locks with write locks. In contrast, TxnSails employs validation-based concurrency control in a fine-grained manner, achieving superior performance compared to state-of-the-art approaches. Furthermore, unlike previous work that merely advocates for a lower isolation level, we argue that, due to the varying structures and proportions of different transaction templates, higher isolation levels can sometimes yield better results, which can be captured and used by TxnSails adaptively.

8 RELATED WORK

Our study is related to the previous work on concurrency control algorithms that ensure serializable scheduling within and outside the database kernel.

Within the database kernel. Existing works have explored a variety of algorithms to guarantee SER, which can be divided into two main categories: (1) deterministic algorithms (e.g., Calvin [52], Aria [45], Harmony [39]) and (2) non-deterministic algorithms, such as 2PL, OCC, timestamp ordering (TO) and their variants [16, 17, 38, 41, 53, 57–59, 63–65]. While these algorithms effectively eliminate anomalies in concurrent transaction execution, they offer varying performance benefits depending on the workload. To address this, some studies propose adaptive concurrency control algorithms for dynamic workloads. For instance, SMF [22] greedily selects the next transaction based on the one that would result in the least increase in execution time. Tebaldi [49] constructs a hierarchical concurrency control model by analyzing stored procedures.Polyjuice [56] employs reinforcement learning to design tailored concurrency control mechanisms for each stored procedure, while Snapper [42] mixes deterministic and non-deterministic (i.e., 2PL) algorithms. However, deterministic algorithms impose stricter requirements than TxnSails: (1) transactions must be precollected and completed in a single interactive round, and (2) their read-write sets must be known in advance. Furthermore, all these algorithms are explicitly tailored for database kernels, which limits their broader applicability and generalizability. In contrast, TxnSails requires no kernel modifications and integrates seamlessly with various database systems. More importantly, TxnSails boosts performance by adaptively assigning the optimal isolation level based on workload characteristics while preserving SER.

Outside the database kernel. Substantial efforts have been dedicated to enhancing transaction management in database middleware, such as Tuxedo, Encina, CICS, and ShardingSphere. These systems primarily ensure reliable transaction execution and maintain atomicity and consistency across distributed databases, relying on the underlying databases for concurrency control. Their contributions are orthogonal to those of TxnSails, which focuses on serializable scheduling and self-adaptive isolation level selection. Importantly, our techniques can be integrated into these middleware systems to further enhance their performance [4]. Some application developers prioritize application-level concurrency control using mechanisms such as Java's ReentrantLock or memory stores like Redis [1]. Tang et al. [50, 51, 60] provide valuable insights into ad-hoc transactions, which offer flexible and efficient concurrency control on the application side. Bailis et al. [14] introduce the application-dependent correctness criterion known as *I-confluence*, which evaluates whether coordination-free execution preserves application invariants. Conway et al. [23] use monotonicity analysis to eliminate the need for coordination in distributed applications. These techniques demand that developers have a high level of expertise in concurrency control, which can increase the risk of errors. In contrast, TxnSails relieves programmers from the complexities of concurrency control, ensuring efficiency through self-adaptive isolation level selection with minimal application modifications. The most relevant work to ours demonstrates that scheduling entire workloads under low isolation levels can still achieve SER by adjusting specific query patterns. Fekete et al. provide the necessary and sufficient conditions for SI to achieve serializable scheduling [10, 27]. Ketsman et al. [37, 54] investigate the characteristics of non-serializable scheduling under RC and Read Uncommitted isolation levels. This theoretical framework has been further refined with functional constraints by Vandevoort et al. [55]. Based on these insights, TxnSails can accurately and efficiently achieve SER across various isolation levels. To the best of our knowledge, TxnSails is the first work to model the trade-off between the performance benefits and the serializability overhead under low isolation levels, achieving the self-adaptive isolation level selection.

9 CONCLUSION

In this paper, we present TxnSails, an efficient middle-tier approach that achieves serializability by strategically selecting between serializable and low isolation levels for dynamic workloads. TxnSails introduces a unified middle-tier validation method to enforce the commit order consistent with the vulnerable dependency order, ensuring serializability in single-isolation and cross-isolation scenarios. Moreover, TxnSails adopts a graph-learned model to extract the runtime workload characteristics and adaptively predict the optimal isolation levels, achieving further performance improvement. The experimental results show that TxnSails can self-adaptively select the optimal isolation level and outperform state-of-the-art solutions and the native PostgreSQL.

REFERENCES

[1] 2024. Redis - The Real-time Data Platform. https://redis.io/.
[2] 2024. Supplemental materials of TxnSails. https://github.com/dbiir/TxnSailsServer.
[3] 2024. t-distributed stochastic neighbor embedding. https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding.
[4] 2025. Integrate TxnSails into Apache Shardingsphere. git@github.com:dbiir/GeoTP_middleware.git.
[5] 2025. Neon Serverless Postgres — Ship faster. https://neon.tech/.
[6] 2025. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.
[7] 2025. TPC-C: On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/.
[8] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *ICDE*. IEEE Computer Society, 67–78.
[9] Mohammad Alomari et al. 2009. Ensuring serializable executions with snapshot isolation dbms. (2009).
[10] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Röhm. 2008. Serializable executions with snapshot isolation: Modifying application code or mixing isolation levels?. In *Database Systems for Advanced Applications: 13th International Conference, DASFAA 2008, New Delhi, India, March 19-21, 2008. Proceedings 13*. Springer, 267–281.
[11] Mohammad Alomari, Michael J. Cahill, Alan D. Fekete, and Uwe Röhm. 2008. The Cost of Serializability on Platforms That Use Snapshot Isolation. In *ICDE*. IEEE Computer Society, 576–585.
[12] Mohammad Alomari and Alan D. Fekete. 2015. Serializable use of Read Committed isolation level. In *AICCSA*. IEEE Computer Society, 1–8.

[13] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3 (2013), 181–192.

[14] Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (2014), 185–196.

[15] Peter Bailis, Alan D. Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable atomic visibility with RAMP transactions. In *SIGMOD Conference*. ACM, 27–38.

[16] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (2019), 2325–2338.

[17] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[18] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *ICLR*.

[19] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD Conference*. ACM, 729–738.

[20] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable isolation for snapshot databases. *ACM Trans. Database Syst.* 34, 4 (2009), 20:1–20:42.

[21] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: Tencent Distributed Database System. *Proc. VLDB Endow.* 17, 12 (2024), 3869–3882.

[22] Audrey Cheng, Aaron N. Kabcenell, Jason Chan, Xiao Shi, Peter D. Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards Optimal Transaction Scheduling. *Proc. VLDB Endow.* 17, 11 (2024), 2694–2707.

[23] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *SoCC*. ACM, 1.

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. ACM, 143–154.

[25] Akon Dey, Alan D. Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE Workshops*. IEEE Computer Society, 223–230.

[26] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[27] Alan D. Fekete. 2005. Allocating isolation levels to transactions. In *PODS*. ACM, 206–215.

[28] Alan D. Fekete. 2019. Making Consistency Protocols Serializable. In *PODS*. ACM, 269.

[29] Alan D. Fekete, Shirley Goldrei, and Jorge Perez Asenjo. 2009. Quantifying Isolation Anomalies. *Proc. VLDB Endow.* 2, 1 (2009), 467–478.

[30] Alan D. Fekete, Elizabeth J. O'Neil, and Patrick E. O'Neil. 2004. A Read-Only Transaction Anomaly Under Snapshot Isolation. *SIGMOD Rec.* 33, 3 (2004), 12–14.

[31] Satoshi Furutani, Toshiki Shibahara, Mitsuaki Akiyama, Kunio Hato, and Masaki Aida. 2019. Graph Signal Processing for Directed Graphs Based on the Hermitian Laplacian. In *ECML/PKDD (1) (Lecture Notes in Computer Science)*, Vol. 11906. Springer, 447–463.

[32] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 11 (2020), 2773–2786.

[33] Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. 1995. *Bayesian data analysis*. Chapman and Hall/CRC.

[34] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *ICML (Proceedings of Machine Learning Research)*, Vol. 70. PMLR, 1263–1272.

[35] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564.

[36] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (2017), 553–564.

[37] Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. 2022. Deciding Robustness for Lower SQL Isolation Levels. *ACM Trans. Database Syst.* 47, 4 (2022), 13:1–13:41.

[38] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD Conference*. ACM, 1675–1687.

[39] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When Private Blockchain Meets Deterministic Database. *Proc. ACM Manag. Data* 1, 1 (2023), 98:1–98:28. https://doi.org/10.1145/3588952

[40] Ruiyuan Li, Liang Zhang, Juan Pan, Junwen Liu, Peng Wang, Nianjun Sun, Shanmin Wang, Chao Chen, Fuqiang Gu, and Songtao Guo. 2022. Apache ShardingSphere: A Holistic and Pluggable Platform for Data Sharding. In *ICDE*. IEEE, 2468–2480.

[41] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD Conference*. ACM, 21–35.

[42] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2022. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 65–78.

[43] David B. Lomet. 1993. Key Range Locking Strategies for Improved Concurrency. In *VLDB*. Morgan Kaufmann, 655–664.

[44] David B. Lomet, Alan D. Fekete, Rui Wang, and Peter Ward. 2012. Multi-version Concurrency via Timestamp Range Conflict Management. In *ICDE*. IEEE Computer Society, 714–725.

[45] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 11 (2020), 2047–2060. http://www.vldb.org/pvldb/vol13/p2047-lu.pdf

[46] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *SIGMOD Conference*. ACM, 631–645.

[47] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[48] Dan R. K. Ports and Kevin Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.

[49] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 283–297.

[50] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *SIGMOD Conference*. ACM, 4–18.

[51] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2023. Ad Hoc Transactions: What They Are and Why We Should Care. *SIGMOD Rec.* 52, 1 (2023), 7–15.

[52] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*. ACM, 1–12.

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.

[54] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2021. Robustness against Read Committed for Transaction Templates. *Proc. VLDB Endow.* 14, 11 (2021), 2141–2153.

[55] Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. 2022. Robustness Against Read Committed for Transaction Templates with Functional Constraints. In *ICDT (LIPIcs)*, Vol. 220. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:17.

[56] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. 198–216.

[57] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2015. The Serial Safety Net: Efficient Concurrency Control on Modern Hardware. In *Proceedings of the 11th International Workshop on Data Management on New Hardware* (Melbourne, VIC, Australia). Association for Computing Machinery, New York, NY, USA, Article 8, 8 pages.

[58] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 26, 4 (2017), 537–562.

[59] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2018. Erratum to: Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 27, 6 (2018), 899–900.

[60] Zhaoguo Wang, Chuzhe Tang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2024. Ad Hoc Transactions through the Looking Glass: An Empirical Study of Application-Level Transactions in Web Applications. *ACM Trans. Database Syst.* 49, 1 (2024), 3:1–3:43.

[61] Tao Yu, Zhaonian Zou, Weihua Sun, and Yu Yan. 2024. Refactoring Index Tuning Process with Benefit Estimation. *Proc. VLDB Endow.* 17, 7 (2024), 1528–1541.

[62] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220.

[63] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD Conference*. ACM, 1629–1642.

[64] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10

(2018), 1289–1302.
[65] Zhanhao Zhao, Hongyao Zhao, Qiyu Zhuang, Wei Lu, Haixiang Li, Meihui Zhang, Anqun Pan, and Xiaoyong Du. 2023. Efficiently Supporting Multi-Level Serializability in Decentralized Database Systems. *IEEE Trans. Knowl. Data Eng.* 35, 12 (2023), 12618–12633.
[66] Qiushi Zheng, Zhanhao Zhao, Wei Lu, Chang Yao, Yuxing Chen, Anqun Pan, and Xiaoyong Du. 2024. Lion: Minimizing Distributed Transactions Through Adaptive Replica Provision. In *ICDE*. IEEE, 2012–2025.
[67] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428.
[68] Zeheng Zhou, Ying Jiang, Weifeng Liu, Ruifan Wu, Zerong Li, and Wenchao Guan. 2024. A Fast Algorithm for Estimating Two-Dimensional Sample Entropy Based on an Upper Confidence Bound and Monte Carlo Sampling. *Entropy* 26, 2 (2024), 155.